Omówienie zadań z Turnieju 24 (dla początkujących)

Karol Pokorski

1 Kamień, papier, nożyce

W tym zadaniu należało po prostu wprost zaimplementować warunki gry w kamień, papier i nożyce.

W sekcji *Wejście* opisano, że należy wczytać dwie linie, w każdej z nich będzie jedno słowo (albo `KAMIEN`, albo `PAPIER`, albo `NOZYCE` – podane było wielkimi literami oraz bez polskich znaków i tak należało założyć). Pierwsze słowo oznacza symbol pokazany przez Jasia, a drugie słowo – symbol pokazany przez Małgosię. Słowa najlepiej wczytać do zmiennej typu `string` (w przypadku C++), albo typu `str` (w przypadku Pythona).

Jest tylko 9 różnych możliwości wejść do programu (trzy możliwości na symbol Jasia razy trzy możliwości na symbol Małgosi). Możliwe jest przygotowanie dziewięciu `if`ów.

Ale możliwe jest też uproszczenie sobie zadania i sprawdzenie najpierw czy symbole Jasia i Małgosi są sobie równe (czyli czy wczytane napisy na wejściu są równe) i wypisanie `REMIS` w takiej sytuacji i zakończenie programu (w C++ można do tego użyć `return 0;` w funkcji `main()` lub `exit(0);` gdziekolwiek indziej). Jednym sprawdzeniem eliminujemy więc trzy możliwe wejścia, zostało jeszcze sześć (trzy z nich faworyzują Jasia, a pozostałe Małgosię). Można więc teraz sprawdzić trzy warunki, w których wygrywa Jasio (jeden z nich to `if (jasio == "KAMIEN" && malgosia == "NOZYCE")`, są jeszcze dwa inne). Jeżeli wejście programu do każdego z tych warunków powoduje wypisanie napisu `JASIO` **oraz zakończenie programu**, a program dalej się jeszcze wykonuje (już po sprawdzeniu tych warunków), to możemy bezpiecznie wypisać `MALGOSIA`.

2 Suma cyfr sumy cyfr

W zadaniu tym trzeba było rozwiązać równanie $s(s(x)) = n$, dla znanej wartości n oraz funkcji $s : \mathbb{N}_+ \to \mathbb{N}_+$, zwracającej sumę cyfr. Dokładniej, trzeba było znaleźć najmniejsze rozwiązanie naturalne x, że suma cyfr sumy cyfr x'a będzie równa n.

Myśląc nieco matematycznie, przydałoby się stworzyć funkcję odwrotną do funkcji sumy cyfr (nazwijmy tę funkcję s^{-1}): mając zadane ustaloną sumę cyfr, powinna ona zwracać najmniejszą liczbę o tej sumie cyfr (czyli na razie będzie to funkcja, która „cofa" o jedną aplikację funkcji $s()$, a nie dwie).

Nie jest trudno zgadnąć, jak obliczyć funkcję $s^{-1}(n)$:

- jeżeli $n \leqslant 9$, wystarczy zwrócić n,
- jeżeli $n \geqslant 10$, wystarczy napisać na końcu liczby dziewiątkę, a przed tym napisać $s^{-1}(n-9)$. Innymi słowy wystarczy zwrócić $10s^{-1}(n-9)+9$.

To działa, bo:

- liczba cyfr jest minimalizowana,
- oraz (w miarę możliwości) pierwsza cyfra jest najmniejsza możliwa.

Teraz kluczowa obserwacja: funkcja $s^{-1}(n)$ jest rosnąca: im większe n, tym większa musi być liczba o tej sumie cyfr. A to oznacza, że żeby odwrócić $s(s(x)) = n$, wystarczy tylko wypisać $x = s^{-1}(s^{-1}(n))$. Innymi słowy: nie opłaca się wygenerować z zadanej sumy cyfr większej liczby niż najmniejsza, licząc na to że po drugim zaaplikowaniu do $s^{-1}(\cdot)$ otrzymamy coś mniejszego.

Drobnym problemem jest fakt, że dla $n = 60$, wynik jest bardzo długi i zdecydowanie nie mieści się w zmiennej typu `long long int`. Warto więc przygotować dwa warianty funkcji s^{-1}:

- jeden na pierwsze uruchomienie, który zwraca odpowiednią liczbę w postaci liczby (będzie ona co najwyżej 6 999 999, a więc zmieści się nawet w zmiennej typu `int`),
- drugi na drugie uruchomienie, który zwróci odpowiedni napis składający się z $\lfloor \frac{n}{9} \rfloor$ dziewiątek oraz reszty z dzielenia n przez 9 na początku (trzeba uważać na ewentualne zero wiodące, żeby go nie wypisać).

3 Liczba nawiasowań

Pewna wskazówka jak rozwiązać to zadanie, znajdowała się w treści zadania. Podano tam, że liczba nawiasowań długości $2n$ z jednym typem nawiasów jest równa n-tej liczbie Catalana.

Weźmy dowolne poprawne nawiasowanie z użyciem jednego typu nawiasów. Dla każdej pary odpowiadających sobie nawiasów, możemy ją zamienić na jeden z trzech typów nawiasów. Każdy z tych typów nawiasów można ustalić niezależnie, a więc z jednego poprawnego standardowego nawiasowania otrzymujemy 3^n różnych nawiasowań z użyciem trzech typów nawiasów.

Co więcej, jesteśmy tak w stanie uzyskać każde z nawiasowań z użyciem trzech nawiasowań i to dokładnie na jeden sposób. To oznacza, że wynikiem zadania dla podanego na wejściu n jest po prostu $3^n \cdot C_n$.

Pozostaje kwestia obliczenia wyniku modulo $10^9 + 7$. W przypadku potęgi trójki jest to łatwe: wystarczy domnażać trójkę n razy i z każdym domnożeniem wykonać na wyniku operację modulo $10^9 + 7$ (trzeba być ostrożnym na przekroczenie zakresu `int`a, chociaż jeden miliard mieści się w tej zmiennej, to już trzy miliardy nie mieszczą się, lepiej dla bezpieczeństwa użyć zmiennej typu `long long int`). Nieco gorzej jest z n-tą liczbą Catalana. W większości wzorów (podanych w treści zadania) występuje dzielenie, z którym trudno sobie poradzić (jest to co prawda możliwe, wystarczy użyć operacji odwrotności modularnej). Limity w zadaniu były jednak na tyle małe, że możliwe było zastosowanie wzoru, w którym nie ma dzielenia (np. obliczenie symbolu Newtona z trójkąta Pascala lub obliczenie liczby Catalana z wzoru rekurencyjnego, czwartego na liście wzorów podanych w zadaniu).

4 Oś liczbowa

Jest to typowe zadanie implementacyjne. Niby wiadomo jak je rozwiązać, ale doprowadzenie szczegółów do porządku wydaje się nie być takie łatwe.

Zacznijmy od wczytania wejścia. Najłatwiej chyba wczytać pierwszy wiersz jako napis, zliczyć liczbę znaków `+`, a następnie wczytywać drugi wiersz wejścia jako kolejne liczby (zwykłym `std::cin`, który zignoruje nadmiarowe spacje). Dobrym pomysłem jest stworzyć sobie mapowanie (np. w postaci tablicy par, dwóch osobnych tablic lub słownika typu `std::map`) z pozycji znaku plus w napisie na odpowiadającą jej liczbę na osi (z drugiego wiersza wejścia).

Mając to, można zacząć wczytywać kolejne (dodawane na oś) punkty. Należy je dodać (nie ma znaczenia w jakiej kolejności) do wcześniej stworzonego mapowania: czyli konieczne jest przeliczenie wartości liczby na pozycję plusa na osi (w pierwszym wierszu wejścia). Nie jest to trudne: skalę osi można wyznaczyć na podstawie dwóch dowolnych wcześniej zaznaczonych punktów (jako iloraz różnicy wartości przypisanych punktów i różnicy pozycji plusów – należy uważać, bo może być to wynik

niecałkowity). Dodatkowe plusy można od razu zaznaczyć w zmiennej, której użyło się do wczytania pierwszego wiersza wejścia.

Ostatnia trudność to umieszczenie etykiet z liczbami w drugim wierszu wejścia. Można stworzyć napis, początkowo składający się z samych spacji (o długości takiej jak oś) i „wprasowywać" w niego wszystkie wartości z utworzonego mapowania (starych i nowych punktów). Dla każdego takiego punktu dobrze jest skonwertować zapisywaną wartość (etykietę na osi) do napisu (w C++ służy do tego funkcja `to_string`, a w Pythonie konstruktor `str`). Znając pozycję plusa oraz długość liczby i mając zapisane liczbę w postaci napisu, łatwo już przepisać odpowiednie znaki tam gdzie trzeba.

5 Świat według Fibonacciego

Jakimś pomysłem na (częściowe) rozwiązanie zadania jest sprawdzić każdą liczbę M od 1 do $X-1$ włącznie. Najpierw wyznaczamy tablicę liczb Fibonacciego (można ją umieścić w jakimś zbiorze, np. `std::set`, który pozwala szybko sprawdzać czy jakiś element do niego należy czy nie). Liczymy resztę z dzielenia X przez M i sprawdzamy czy jest we wcześniej obliczonym zbiorze liczb Fibonacciego. Trudno jest jednak ten pomysł zoptymalizować, żeby mieć szansę zaliczyć wszystkie testy. Dla wartości X powyżej kilku milionów program będzie po prostu za wolny.

Lepiej chwilę pomyśleć: liczb Fibonacciego poniżej X (możliwych interesujących nas wyników obliczenia reszty z dzielenia X przez M) jest niewiele, możemy więc ustalić sobie konkretną liczbę F_k i zacząć się zastanawiać, ile jest liczb M, dla których $X \bmod M = F_k$. Jeżeli X ma dawać resztę F_k przy dzieleniu przez M, to różnica $X - F_k$ musi dawać resztę 0 przy dzieleniu przez M. Jeszcze inaczej mówiąc: M musi być dzielnikiem różnicy $X - F_k$ (większym niż F_k). Algorytmem „do pierwiastka" można wyznaczać dzielniki dowolnej liczby Z (dla każdego małego dzielnika p jest odpowiadający mu duży dzielnik q do pary, tak żeby $p \cdot q = Z$, czyli $q = \frac{Z}{p}$).

Podsumowując, do rozwiązania zadania wystarczy trochę sprytu, wyznaczenie tablicy liczb Fibonacciego, funkcja obliczająca zbiór dzielników liczby, wywołanie tej funkcji dla każdej różnicy $X - F_k$ i zliczenie liczby tych dzielników, które są większe niż F_k.

6 Kąty wielokąta foremnego

To zadanie celowo było podane jako szóste w kolejności, aby zachęcić zawodników do wspomagania się rankingiem, w celu znajdowania łatwych zadań. Nie zawsze bowiem jest tak (a wręcz jest tak rzadko), że zadania są uporządkowane w kolejności od najłatwiejszego do najtrudniejszego.

Zadanie można streścić następująco. Dana jest miara kąta wielokąta foremnego w stopniach i należy powiedzieć jaki to jest wielokąt (ile ma boków).

Możliwe jest rozwiązanie tego zadania z użyciem nietrudnej matematyki: suma miar kątów wewnętrznych w dowolnym n-kącie jest równa $180(n-2)^\circ$ (uogólnienie twierdzenia o sumie miar kątów w trójkącie). Ponieważ w wielokącie foremnym wszystkie kąty mają równą miarę, otrzymujemy wzór na miarę kąta $\alpha = \frac{180(n-2)}{n}^\circ$. Teraz możliwe jest rozwiązanie tego równania ze względu na n (przekształcenie tego wzoru): $n = \frac{360}{180-\alpha}$ lub wręcz zgadnięcie n iterując po kolejnych wartościach od 1 do 360 i sprawdzając czy z pierwszego równania wychodzi nam wczytany wynik α.

Należało jedynie uważać na:

- ewentualne błędy zaokrągleń (w skutek których może się nam wydawać, że dane n jest poprawne, chociaż uzyskany kąt różni się od poprawnego, ale o mniej niż 1°; lub w których obliczone n zamiast być równe np. 45, będzie równe $44.99999\ldots$ i po zaokrągleniu w dół otrzymamy 44),
- dzielenie przez 0 (przy kątach typu 0°, 180° lub 360°, dla których odpowiedź jest oczywiście `NIE`, ale nie zwalnia to z odpowiedzialności za poprawne zakończenie programu bez błędu).

7 Zadanie dla dzieci

To było najtrudniejsze zadanie tego turnieju.

Kluczowa (i chyba jedyna) obserwacja w tym zadaniu to uświadomienie sobie, że każdą konfigurację monet w zadaniu można opisać jedną liczbą: liczbą reszek. Kolejność monet nie ma bowiem żadnego znaczenia, a liczbę orłów można trywialnie uzyskać odejmując od liczby wszystkich monet liczbę reszek.

A skoro N jest co najwyżej 5 000, to różnych stanów w grze jest co najwyżej 5 001 i może stać nas na to, żeby przeanalizować każdy z nich.

Do jakich stanów możemy przejść ze stanu X reszek? Za każdym razem musimy przewrócić na drugą stronę dokładnie K monet. Jeżeli będą to same reszki i przewrócimy je na orły to bilans będzie $-K$ (o K reszek mniej niż było wcześniej). Jeżeli będą to prawie same reszki i jeden orzeł to bilans będzie $-K + 2$. Prowadząc takie rozważanie, dochodzimy do wniosku, że potencjalnie możliwe zmiany liczby reszek z dowolnego stanu są o jedną z wartości ze zbioru $\{-K, -K+2, -K+4, \ldots, K-4, K-2, K\}$. Nie z każdej liczby reszek możliwe jest jednak przejście o każdą wartość z tego zbioru. Nie jest możliwe zabranie więcej reszek/orłów niż jest w danej chwili. Ustalenie odpowiedniego podzbioru z każdego stanu X jest jednak kwestią dopisania do programu zaledwie kilku warunków. Niech $\delta(X)$ oznacza zbiór wszystkich stanów, do których można dojść w jednym ruchu ze stanu X.

Pora zacząć rozwiązywać zadanie. Dobrym pomysłem jest utrzymywać listę wszystkich stanów X, do których można dojść optymalnie z użyciem każdej liczby ruchów. Na początku mamy listę stanów dla 0 ruchów: jest to jedynie $\{X\}$. Lista stanów dla jednego ruchu jest równa zbiorowi $\delta(X)$. Aby uzyskać wszystkie stany dla $k + 1$ ruchów, można przeanalizować zbiór $\delta(\cdot)$, dla każdego elementu zbioru dla k ruchów. Ze względów wydajnościowych, chcemy przy tym ignorować stany, do których można było dojść z mniejszą liczbą ruchów.

Bardziej spostrzegawczy i zaznajomieni zawodnicy mogą tutaj zauważyć (nieprzypadkową) analogię do algorytmu grafowego BFS. Istotnie: jeżeli założyć, że mamy graf, w którym wierzchołkami są liczby reszek X, a krawędzie prowadzą do elementów zbiorów $\delta(\cdot)$, to zadanie sprowadza się do znalezienia najkrótszej ścieżki w grafie (które robi się w zasadzie dokładnie tak samo, jak opisano w akapicie powyżej).